Vérification formelle des communications dans un réseau sur puce : HERMES à l'aide d'ACL2

Amr Helmy, Laboratoire TIMA - 46 Av. Félix Viallet - 38031 Grenoble cedex 1, amr.helmy@imag.fr

Résumé- **Pour la conception des systèmes multi-composants complexes (SoCs), une technique largement répandue aujourd'hui est la réutilisation de modules préfabriqués (IPs) interconnectés grâce à un réseau sur puce (NoC). Ce papier s'intéresse à la vérification formelle de réseaux sur puces à l'aide d'un démonstrateur de théorèmes automatique (l'outil ACL2). Dans une précédente thèse, un modèle formel générique pour les NoCs a été proposé et mis en œuvre dans ACL2, ce papier propose quelques extensions pour ce modèle et décrit la preuve d'un réseau spécifique, HERMES.**

I. INTRODUCTION

La vérification de correction des circuits avant commercialisation est une phase primordiale. Un exemple frappant de ces dernières années est le bug découvert sur le Pentium d'Intel après sa commercialisation qui a coûté à l'entreprise 400 millions de dollars [1]. Avec l'augmentation de la complexité des systèmes, de plus en plus de temps est consacré à la vérification (60% à 70% [2]). Avec l'utilisation des composants préfabriqués (IPs), le travail de vérification consiste essentiellement en deux parties :

- La vérification des IPs.
- La vérification du réseau de connexion.

La vérification des IPs étant faite en amont lors de leur conception, le problème essentiel reste la vérification du réseau d'interconnexion des composants.

L'interconnexion peut entraîner alors l'apparition de plusieurs problèmes classiques qu'il s'agira de maîtriser lors de la conception du circuit afin de ne pas entraîner d'erreurs via un mauvais fonctionnement de la communication entre les composants. Ces principales difficultés liées aux réseaux et plus particulièrement aux réseaux sur puces (Network On Chip, ou NoCs) sont :

- « Deadlock » : dépendance cyclique entre les nœuds à la requête de la même ressource.
- « Livelock » : paquets tournant dans le réseau sans progrès vers la destination.
- « Starvation » : un paquet bloqué dans un buffer indéfiniment, car la sortie n'est pas disponible.

La solution consiste à vérifier les algorithmes de routage et d'ordonnancement et leur implémentation dans le réseau. Notamment, un réseau est souvent divisé logiquement en deux parties :

1. Les services : dépendent des protocoles implémentés
2. Le système de communication : responsable du transfert d'information entre la source et la destination.

Les services étant implémentés dans les différents composants il ne reste essentiellement qu'à vérifier le système de communication. Si le processus de vérification peut s'abstraire de la taille des données, des bus et du réseau, le système de communication est vérifié indépendamment des valeurs des paramètres du système sur puce.

Le travail présenté ici se place dans le contexte de la vérification formelle des réseaux sur puces. Un modèle générique a été déjà présenté, baptisé GeNoC [3] . Nous avons réalisé une extension à ce modèle et son application à un réseau réel : HERMES [4] .

L'outil de démonstration automatique dans lequel GeNoC a été implémenté est le prouveur ACL2 [5] [6] . Il est basé sur la logique du premier ordre, sans quantificateur, et avec égalité.

ACL2 utilise un sous-ensemble applicatif de Common Lisp : un langage qui manipule des expressions préfixées.

Les modèles écrits en ACL2 peuvent être exécutés avec une vitesse comparable à celle des programmes écrits en C.

Cet outil a été déjà utilisé pour la vérification de plusieurs types de composants : microprocesseurs [7], unité virgule flottante [8] , et d'autres structures [9].

La section II présente les principes de base des réseaux sur puces, ainsi que du réseau HERMES. La section III contient la spécification de GeNoC. La section IV décrit les extensions apportées au modèle et la validation d'HERMES. La section V illustre la simulation de ce modèle, et la conclusion est donnée dans la section VI.

II. RESEAUX SUR PUCE ET HERMES

Afin d'optimiser la phase de production lors de la création de systèmes multi composants (System On Chip, ou SoC), une technique développée largement consiste à créer des modèles réutilisables. En particulier, les modèles de réseaux sur puces (NoC) permettent à l'ingénieur de s'affranchir de la complexité liée à la création d'un réseau en reprenant un modèle correspondant à ses attentes.

Les réseaux sur puce reposent sur les mêmes concepts que les réseaux généraux. Néanmoins il existe deux problèmes majeurs :

1. Les systèmes embarqués fonctionnent en temps réel.
2. Les systèmes embarqués fonctionnent sous de fortes contraintes d'énergie et de puissance.

Un nœud est formé généralement d'un processeur, d'une unité de mémoire, et d'autres fonctionnalités. Les messages sont transmis à l'aide d'un pont (routeur). Les topologies des réseaux les plus répandues sont les grilles, les cubes et les tores.

Les NoCs font l'objet de divers travaux de recherche, touchant à la topologie, au dimensionnement, à l'optimisation des performances, etc.…[10] [11] [12] [13]. Le réseau HERMES est l'un des résultats de ces recherches [14][15] . Le réseau HERMES est le fruit des travaux de Fernando Moraes, Ney Calazans, Aline Mello, Leandro M.Oller, et Luciano Ost, de l'université catholique de Rio Grande do Sul avec la collaboration du Laboratoire d'Informatique de Robotique et de Microélectronique de Montpellier. Ce réseau fut développé d'une part pour répondre au problème posé par l'utilisation des NoCs dans les systèmes intégrés et d'autre part pour apporter une contribution aux divers travaux de recherches menés dans plusieurs autres groupes de chercheurs.

Ce réseau a la forme de grille à deux dimensions (Fig.1). Chaque IP est couplée à un routeur.

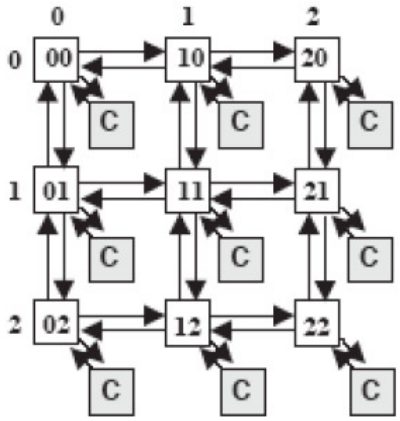

Fig.1, La grille de HERMES

Chaque routeur contient cinq ports bidirectionnels (1 pour l'application, 4 pour les autres routeurs), chaque port ayant un buffer d'entrée (Fig.2). La communication entre les routeurs est asynchrone, alors que dans les routeurs les transactions sont synchrones (Globalement Asynchrone Localement Synchrone).

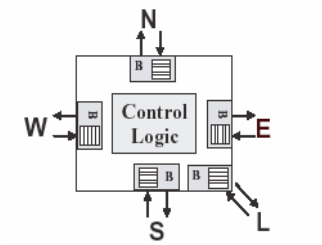

Fig.2, routeur de HERMES

Le routeur est divisé en trois parties:

1. La logique de routage.
2. La logique d'arbitrage.
3. Les ports de communication.

Les messages arrivent sur les ports d'entrées alors une requête est envoyée à la logique d'arbitrage. Pour choisir le port d'entrée à servir, cette logique utilise un arbitrage à priorité tournante (round-robin) pour organiser l'accès aux ports. Puis, il passe une requête à la logique de routage en indiquant le port par lequel le message à router est arrivé.

Un routage XY est utilisé : les messages se déplacent d'abord le long de l'axe des X, puis le long de l'axe des Y.

La commutation par ver de terre (Wormhole) [16] a été choisie car elle offre une faible latence, un faible demande de mémoire, et le canal physique peut être divisé en plusieurs canaux logiques. Dans l'ordonnancement par ver de terre, chaque message est divisé en plusieurs morceaux appelés « flits » dont la taille est paramétrable. Seul le premier flit est routé et les autres flits le suivent. Pour indiquer la fin du message, deux solutions existent. La première est d'avoir un flit EOM (End Of Message) à la fin du message. La deuxième consiste à avoir un flit contenant le nombre de flit constituants le message. Le réseau HERMES utilise la deuxième solution.

Les flits reçus ou bloqués sont stockés dans un buffer (ceci limite la chute des performances). Le stockage des messages affecte tout le réseau. Des buffers à FIFO paramétrable sont utilisés pour chaque port d'entrée.

III. LE MODELE GENOC

Le modèle GeNoC [3] [17] s'appuie sur une représentation abstraite des communications (Fig.3). Il faudra noter que c'est le premier modèle formalisant les réseaux.

Chaque nœud est séparé en une application et une interface. Les interfaces sont connectées à l'architecture de communication. Applications et interfaces communiquent à l'aide des *messages*, tandis que la communication entre les nœuds se fait au moyen de *trames (frames)*.

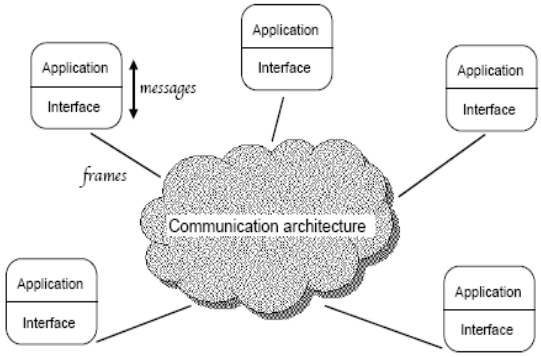

Fig.3, Abstraction de GeNoC

Voyons les définitions de quelques termes employés dans la suite :

Transactions : Une opération de communication est modélisée par une transaction qui précise l'identificateur du message à envoyer, le *message*, la source, et la destination. .

Missives : une *missive* est une *transaction* où le *message* a été converti en *trame*.

Tentatives : Chaque nœud a un nombre de tentatives qu'il peut utiliser pour envoyer les missives, qui décroit à chaque appel de la fonction d'ordonnancement.

Voyages : L'association d'une *trame* à une liste de routes est un *voyage*. C'est le résultat obtenu après le passage d'une *missive* dans une fonction de routage.

Résultats : chaque *voyage* reçu constitue un résultat.

Le modèle générique de communication est formalisé par une fonction baptisée *GeNoC*. Celle-ci suppose que tous les nœuds du réseau sont modélisés selon le même schéma, ainsi que la topologie du réseau et l'algorithme de routage sont quelconques. Cette fonction peut être schématisée par la Fig.4, elle utilise quatre fonctions :

1. P2psend : code le message en trame.
2. P2precv : récupère le message de la trame.
3. Routing : génère la liste des routes pour un message.
4. Scheduling : responsable de l'ordonnancement des messages.

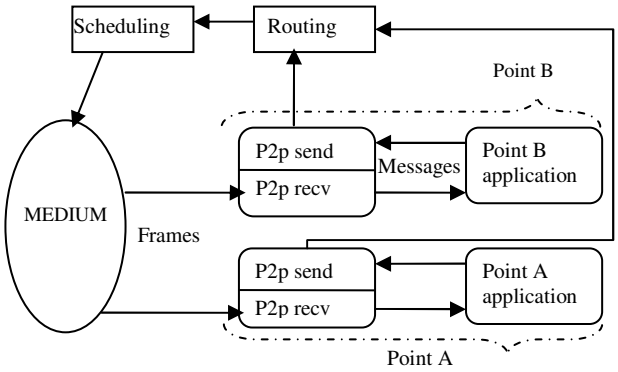

Fig.4, Déroulement de la fonction GeNoC

Le modèle comprend un ensemble de fonctions qui modélisent les dispositifs de l'architecture de communication. Les plus importants expriment la topologie du réseau (le modèle des nœuds et leurs liens), de l'algorithme d'ordonnancement et de routage. La fonction globale, appelée *GeNoC*, prend comme paramètres :

- M : un ensemble de *missives* dans le réseau
- Nodeset : la caractérisation de la structure du réseau,
- att : liste de *tentatives* permises pour envoyer des messages (chaque expéditeur a un nombre maximum de tentatives pour envoyer son message, cette hypothèse est utile pour arrêter la récursion, voir ci-dessous),
- V : l'ensemble des *voyages* qui peuvent être réalisées réellement (à l'origine cette liste est vide).

La fonction *GeNOC* commence avec le test de la condition qu'il reste encore des tentatives si ce la n'est plus le cas, la fonction s'arrête et renvoie une paire de listes, d'un part une liste de *voyages* ordonnancés *Scheduled* et des *voyages* avortés de l'autre *Delayed*. Les *voyages* dans *Delayed* sont transformés de nouveaux en *missives* et passer dans l'appel récursif en tant que *M* (la liste des *missives*).

S'il reste des tentatives, la composition des fonctions *Scheduling* et *Routing* calcule ces deux listes *Scheduled* et *Delayed*.

L'appel récursif de la fonction GeNoC essaie d'ordonnancer les missives dans *M* dès que les *messages* dans *Scheduled* atteignent leur destination et libèrent les nœuds sur leurs chemins.

Définition 1: GeNoC

GeNoC (M, Nodeset, att, V) $\triangleq$
If Sum Of Attempts (att) = 0 **then**
  List (V, M)
Else
  Let (*Scheduled*, *Delayed*, *att1)* **be**
    Scheduling (routing (M, *Nodeset*), att) **in**
    GeNoC (ToMissives (*Delayed*), *Nodeset*, *att1, Scheduled* $\cup$ *V*)
endif

Les fonctions *Scheduling* et *Routing* dans le modèle GeNoC ne sont pas définies. Elles sont associées à des caractéristiques et des obligations de preuve qui sont instanciées pour chaque réalisation d'un réseau donné. Ces théorèmes intermédiaires seront utiles dans la preuve du théorème de correction final.

Le modèle garantit qu'avec la vérification de ces obligations de preuves, le théorème principal de correction est obtenu après la vérification de quelques lemmes intermédiaires :

$\forall\ s \in$ *Scheduled*, $\exists!\ m \in$ *M such that*
Id(s) = Id(m)
$\wedge$ *Frame(s) = Frame(m)*
$\wedge\ \forall\ r \in$ *Routes(s), Last(r) = Dest(m)*

Cet énoncé signifie que le contenu des messages ne change pas et chaque message est reçu par son bon destinataire.

Pour toute instanciation de ce modèle générique dans un cas particulier, il suffit de vérifier les obligations de preuve pour pouvoir obtenir la validité du lemme de correction.

IV. VALIDATION DE HERMES

Le réseau HERMES tel qu'il a été conçu ne rentre pas tout à fait dans le modèle GeNoC. Ceci est dû aux différences entre les spécifications du réseau et les propriétés du modèle:

1. Dans GeNoC les nœuds sont a priori modélisés par leurs coordonnées seulement. Dans HERMES les nœuds sont caractérisés par des coordonnées et 5 ports bidirectionnels.

2. Les messages de HERMES sont découpés en flits, le deuxième flit contenant le nombre de flits qui forment le message. Dans GeNoC il n'existe jusqu'ici aucune caractérisation de la taille du message ni de découpage en flits.

3. Dans le modèle original (chaque nœud n'a implicitement qu'un seul port) un nœud ne peut être utilisé que dans une seule route. HERMES permet à un nœud d'être occupé par plusieurs messages, pourvu qu'ils n'utilisent pas le même port dans la même direction.

Nous allons voir dans la suite comment nous avons pu établir un raisonnement formel sur le réseau HERMES en utilisant la modélisation GeNoC. Les différences évoquées ci-dessus nous ont conduits à fournir certaines adaptations au modèle GeNoC.

A. *Modèle des nœuds*

Dans GeNoC, un nœud de la grille est a priori simplement modélisé comme une paire de coordonnées (x y). Afin de créer un réseau en grille en prenant en compte les ports, nous avons commencé par modéliser les nouveaux nœuds en incluant les ports dans la modélisation (X Y P D) :

- X : la coordonnée sur l'axe des abscisses.
- Y : la coordonnée sur l'axe des ordonnées.
- P : une lettre qui désigne le port, les différentes valeurs pouvant être L pour local, N pour nord, S pour sud, E pour est, W pour ouest.
- D : une lettre désignant la direction du flit (les ports sont bidirectionnels) : « i » pour input, « o » pour output.

La modélisation des nœuds a été obtenue en définissant 10 fonctions. 6 obligations de preuves ont été vérifiées avec un temps total de preuve de 15.9 secondes sur un Intel Pentium 4 sous Linux.

B. *Routage XY*

Dans le modèle GeNoC, comme on l'a vu dans la section III, la fonction GeNoC fait appel à deux autres fonctions : *Routing* et *Scheduling*. Ces deux fonctions forment le corps de tous les calculs de routes et d'ordonnancement.

La fonction *Routing* est responsable de calculer toutes les routes possibles entre deux nœuds. C'est dans cette fonction que l'algorithme modélisant la politique de routage est codé. La fonction prend en entrée une liste de missives, avec les nœuds du réseau, *NodeSet*. Son résultat est une liste de Voyages.

C'est l'instanciation de cette fonction qui permet de modéliser la politique de routage utilisée dans un réseau donné.

Dans le routage XY, on identifie une dimension comme l'axe X et l'autre comme Y. Un paquet est d'abord routé selon l'axe X jusqu'à atteindre l'abscisse de sa destination puis selon l'axe Y pour arriver à la destination [11] [17].

Le routage XY a déjà été modélisé dans [3] . Nous l'avons adapté à la modélisation des nœuds utilisée ici. L'algorithme suivi est celui donné ci-dessous.

Définition 2 : XYRouting

```
XYRouting (from, to) ≜
if from=to                          /* destination atteinte */
        then take the local port
else
    if Xfrom != Xto                 /* change X */
        then if Xfrom < Xto
        then take port East
        else take port West
     else                           /* change Y */
        if Yfrom < Yto
                then take port South
                else take port North
  endif;
```

La fonction du routage a donc été redéfinie pour prendre en compte le nouveau modèle de nœuds décrit en section IV.A, et les trois obligations de preuve associées au routage ont été vérifiées dans ce contexte en passant par 48 théorèmes, pour un temps de preuve de 926.4 secondes.

C. *Ordonnancement par ver de terre*

L'ordonnancement par paquets a été traité dans [3] La modélisation de l'ordonnancement par ver de terre s'en inspire. Une première adaptation concerne le fait qu'un message, en ordonnancement par paquets, libère immédiatement le nœud sur lequel il vient de passer. Dans ce cas, chaque paquet modélise un message et ainsi l'ordonnancement est calculé de façon indépendante sur chacun de ceux-ci. Chaque nœud composant la route des paquets est réservé pour une seule étape.

Comme il a été vu auparavant, la modélisation par verre de terre est effectuée en découpant le message en plusieurs flits. Ainsi l'ordonnancement doit permettre non seulement le passage de l'entête mais aussi de tous les flits consécutifs tout en conservant l'ordre.

L'algorithme doit donc allouer les nœuds qui composeront le chemin que les flits vont emprunter afin de joindre la source et à la destination voulue. Pour cela, certains nœuds sont alloués par le flit de tête, et ceux-ci restent alloués tant que le dernier flit du message ne les a pas traversés.

En ce qui concerne les parties communicantes (source et destination), seul un canal de communication par lequel les flits du message vont être transmis est visible. Ainsi, durant la transmission d'un message, la structure mise en jeu se limite, pour les parties communicantes, à un canal composé des nœuds alloués.

Le réseau HERMES n'utilise pas le modèle du ver de terre avec le flit de terminaison. Le modèle ayant un flit de type NB_FLIT est utilisé.

Dans les missives du modèle GeNoC, quatre champs peuvent être clairement identifiés (Fig. 5) :

1. ID : l'identificateur unique du message.
2. Origine : le nœud émetteur du message.
3. Message : le corps du message envoyé.
4. Destination : le nœud destination du message.

ID	Origine	Message	Destination

Fig. 5, Missives de GeNoC.

Le modèle GeNoC ne contient pas de champ permettant de modéliser le nombre des flits. Deux solutions sont alors possibles :

1. Modéliser chaque flit comme une missive.
2. Ajouter un champ NB_FLIT contenant le nombre de flits.

La première solution ne permettait pas de s'assurer que les flits d'un message allaient obligatoirement se suivre sans séparation et en conservant l'ordre dans lequel ils ont été émis. Nous avons donc retenu la deuxième solution.

Les types de données présentés dans la section III ont donc dû être adaptés.

Transactions : à l'origine les transactions sont formées de quatre champs :

1. IdT : identificateur unique pour chaque transaction.
2. OrgtT : origine du message.
3. MsgT : le contenu à transmettre.
4. DestT : destination du message.

IdT	OrgtT	MsgT	DestT

Fig. 6, transaction de GeNoC

Missives : de même quatre champs forment chaque missive:

1. IdM : Identificateur unique pour chaque missive.
2. OrgtM : origine de la missive.
3. FrmM : le contenu à transmettre.
4. DestM : destination de la missive

IdM	OrgtM	FrmM	DestM

Fig. 7, Missive de GeNoC

Voyages : Trois champs forment le voyage :

1. IdV : identificateur unique pour chaque voyage.
2. FrmV : le contenu à transmettre.
3. RoutesV : la liste des routes retournées par la fonction de routage.

IdV	FrmV	RoutesV

Fig. 8, voyage de GeNoC

Résultats : les résultats sont formés de trois champs :

1. IdR : Identificateur unique pour chaque résultat.
2. DestR : destinataire du message envoyé au début.
3. MsgR : le message envoyé.

IdR	DestR	MsgR

Fig.9, résultat de GeNoC

On s'aperçoit de l'absence de nombre de flits dans tous les types, la modification dans les types sert simplement à ajouter un champ partout pour le nombre de flits.

Les nouveaux types sont décrits dans les figures ci-dessous.

IdT	OrgtT	MsgT	DestT	FlitT

Fig. 10, Nouveau format de Transaction

IdM	OrgtM	FrmM	DestM	FlitM

Fig.11:Nouveau format de missive

IdV	FrmV	RoutesV	FlitV

Fig.12 : Nouveau format de voyage

IdR	DestR	MsgR	FlitR

Fig.13: Nouveau format de résultat

La fonction *Scheduling,* du modèle GeNoC prend comme entrées une liste de voyages, normalement fournis par *Routing,* et une liste de tentatives. La liste de tentatives sert à garantir la terminaison de l'exécution récursive de la fonction.

Cette fonction retourne trois listes, une liste des voyages ordonnancés *Scheduled (S)*, une liste de voyages avortés *Delayed (D)*, et la nouvelle liste de tentatives. Les deux premières listes constituent la sortie fournie par la fonction GeNoC.

Scheduling est responsable de choisir la route que chaque message va suivre. Ceci est fait à l'aide d'un test qui s'assure qu'un message ne peut pas utiliser dans son passage un nœud et un port simultanément avec un message pour lequel le voyage est déjà ordonnancé. Si le voyage est ordonnançable, il est ajouté à la liste *S* sinon il est destiné à la liste *D.*

La fonction aura besoin de connaître les nœuds déjà utilisés. Ceci est fait à l'aide d'une liste interne à cette fonction appelée *Prev (P)* qui modélise l'occupation dans le temps des nœuds du réseau.

Notre version de S*cheduling*, *wormhole-scheduling,* fait appel à *wormhSched* qui est la fonction récursive qui réalise l'essentiel des calculs. Elle prend une liste des voyages pas encore testés *L*, une liste de voyages ordonnancés *S*, une liste des voyages avortés *D*, et la liste de l'utilisation des nœuds *P*. La récursion est faite sur la première liste. Cette fonction utilise la fonction de test d'ordonnançabilité « check_routes », la fonction de mise à jour de la liste des nœuds utilisés « updateP ».

Définition 3 : WormhSched

```
WormhSched (L,S,D,P) ≜
    if empty(L)
        then list(S,D,P)
    else
        let tr=first(L)              /* premier voyage */
        and n=nbflits(tr)            /* nombre de flits */
        and r=routesOf(tr)           /* ensemble de routes */
        and c=check_routes(n,r,P) in
    if c=true then
    WormhSched (rest(L),S ∪ update(tr),
                                 D,updateP(n,P,r))
    else
        WormhSched (rest(L),S,D ∪ tr,P)
    Endif;
```

La fonction de test « *check_routes* » prend pour entrées une route r, le nombre de flits n et la liste *P*, le principe est simple : s'il n'y pas d'intersection entre la route r et les routes de la liste *P* alors *true* est retourné sinon *false*.

Définition 4: check-routes

```
Check-routes (n,r,P) ≜
    If empty (r)
        Then false
    Else If no-intersection(r,n,P)
                Then true
        Else check_routes (rest(r), n, P)
    Endif;
```

Si le résultat du test est vrai, la fonction *wormhSched* adopte le changement dans *P* et ajoute le voyage à la liste *S,* sinon il est ajouté à *D.*

La fonction de mise à jour «*updateP* » est plus compliquée. Elle réalise la mise à jour de la liste P en introduisant la nouvelle occupation de nœuds du réseau par tous les flits du message dont l'ordonnancement vient d'être adopté.

Définition 5: UpdateP

```
UpdateP( n,P,r) ≜
    if n=0
        then update-first-flit(P,r)
    else cons (first (update-first-flit(P,r)),
            updateP(n-1, rest (update-first-flit(P,r)),r)
```

Le résultat final est renvoyé à *wormhSched.* Les quatre obligations de preuve associées à la fonction *scheduling* ont été bien prouvées en passant par 23 théorèmes intermédiaires et 10 fonctions, pour un temps total de 12.3 secondes.

D. Instanciation de GeNoC

Voyons maintenant l'instanciation du modèle à HERMES. La fonction GeNoC est instanciée comme suit :

Définition 6 : GeNoC

```
GeNoC (M, Nodeset, att, V) ≜
If Sum Of Attempts (att) = 0 then
    List (V, M)
Else
   Let (Scheduled, Delayed, att1) be
     wormhSched (XYrouting (M, Nodeset), att) in
     GeNoC (ToMissives (Delayed), Nodeset, att1,
     Scheduled ∪ V)
endif
```

On remarque l'appel de *XYrouting* qui est notre instance de *Routing*. Le résultat est passé à *wormhSched* qui fait les calculs d'ordonnancement et retourne la liste des voyages ordonnancés et la liste de voyages avortés.

Enfin, il nous fallait prouver les obligations de preuve et le lemme de correction: chaque message est reçu par son bon destinataire et le contenu des messages ne change pas.

Ceci est fait en passant par plusieurs lemmes intermédiaires. Il fallait prouver que les listes obtenues sont du type attendu, que les routes suivies par chaque voyage sont bien formées, que les identificateurs des voyages dans la liste *S* sont différents de ceux de la liste *D.*

Les preuves réalisées sont génériques de point de vue de la taille de la grille et du nombre de flits par messages.

Divers travaux ont été menés dans le cadre de la vérification de réseaux ou de protocoles. Certains d'entre eux utilisent des outils de démonstration automatisée (par exemple [19][20][21]), mais se concentrent sur la preuve d'une application donnée et ne proposent pas la formalisation générique réutilisable.

D'autres font plutôt appel a des techniques de type model-checking (par exemple [22][23]) qui ne proposent pas non plus de modèle générique, et ne permettent pas de paramétrer les preuves, par exemple sur la taille de la grille et la longueur des messages.

V. SIMULATION DU MODELE

Un des points forts d'ACL2 est qu'il fournit à la fois un environnement de preuve formelle et d'exécution des programmes (Common Lisp). J Moore démontre dans [18] que de très bonnes performances peuvent être obtenues. En conséquence nos modèles ACL2 sont exécutables. On peut donc simuler le modèle prouvable sur des vraies valeurs. C'est une forme de simulation qui peut servir à s'assurer de la correspondance entre le modèle créé et le système d'origine.

Notre version de HERMES est exécutable, prenons un exemple d'une grille 3x3 avec 3 messages qui transitent sur le réseau (table 1).

Message id	Source	Destination	Nb flits
1	(1 2)	(2 0)	3
2	(1 1)	(2 0)	3
3	(0 1)	(1 2)	5

Table 1 : les messages et leur destinations

Fig. 14 montre la route que chaque message doit prendre pour arriver à sa destination.

Durant un premier cycle de simulation on va obtenir l'ordonnancement pour les messages 1 et 3, durant le deuxième cycle on aura celui du message 2. Ceci est dû à l'intersection entre les routes des messages 1 et 2.

La table 2 ci-dessous donne pas par pas les résultats de simulation, Les flits d'un message sont numérotés dans l'ordre décroissant de nb_flits-1 jusqu'à 0.

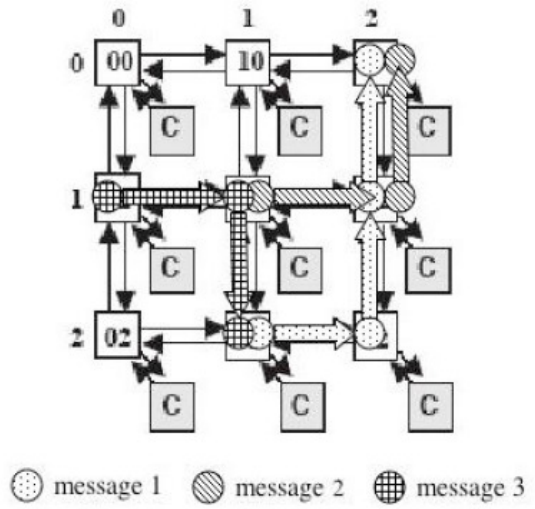

Fig.14, exemple d'exécution

Durant la première étape le flit numéro 4 du message 3 passe à travers le port L dans la direction entrante « i » du nœud (0 1), et le flit 2 du message 1 prend le port L du nœud (1 2) dans la même direction. Durant la deuxième étape, deux flits de chaque message sont visibles dans l'exemple : le flit 4 du message 3 prend le port E du nœud (0 1) dans la direction sortante « o » et le flit 3 prend le port L. Le flit 2 du message 1 prend le port E du nœud (1 2) en direction sortante pendant que le flit 1 entre à travers le port L.

Après 10 étapes, les derniers flits des deux messages arrivent aux ports locaux des nœuds (1 2) et (2 0) dans la direction sortante.

Un outil de visualisation a été implémenté en Java par Yann Dameron. Il permet de visualiser sous forme animée les résultats textuels ci-dessus (voir fig.15).

1ere étape	(((3 4) (0 1 L i)) ((1 2) (1 2 L i)))
2éme étape	(((3 3) (0 1 L i)) ((3 4) (0 1 E o)) ((1 1) (1 2 L i)) ((1 2) (1 2 E o)))
3éme étape	(((3 2) (0 1 L i)) ((3 3) (0 1 E o)) ((3 4) (1 1 W i)) ((1 0) (1 2 L i)) ((1 1) (1 2 E o)) ((1 2) (2 2 W i)))
4éme étape	(((3 1) (0 1 L i)) ((3 2) (0 1 E o)) ((3 3) (1 1 W i)) ((3 4) (1 1 S o)) ((1 0) (1 2 E o)) ((1 1) (2 2 W i)) ((1 2) (2 2 N o)))
5éme étape	(((3 0) (0 1 L i)) ((3 1) (0 1 E o)) ((3 2) (1 1 W i)) ((3 3) (1 1 S o)) ((3 4) (1 2 N i))((1 0) (2 2 W i)) ((1 1) (2 2 N o)) ((1 2) (2 1 S i)))
6éme étape	(((3 0) (0 1 E o)) ((3 1) (1 1 W i)) ((3 2) (1 1 S o)) ((3 3) (1 2 N i)) ((3 4) (1 2 L o))((1 0) (2 2 N o)) ((1 1) (2 1 S i))((1 2) (2 1 N o)))
7éme étape	((3 0) (1 1 W i)) ((3 1) (1 1 S o)) ((3 2) (1 2 N i))((3 3) (1 2 L o)) ((1 0) (2 1 S i)) ((1 1) (2 1 N o)) ((1 2) (2 0 S i)))
8éme étape	((3 0) (1 1 S o)) ((3 1) (1 2 N i)) ((3 2) (1 2 L o)) ((1 0) (2 1 N o)) ((1 1) (2 0 S i))((1 2) (2 0 L o)))
9éme étape	((3 0) (1 2 N i))((3 1) (1 2 L o)) ((1 0) (2 0 S i)) ((1 1) (2 0 L o)))
10éme étape	((3 0) (1 2 L o)) ((1 0) (2 0 L o))

Table 2 : Résultat de la simulation

VI. CONCLUSION

Le modèle GeNoC a été utilisé pour modéliser un réseau ayant des propriétés qui n'avait pas été prises en compte auparavant.

L'idée est d'avoir certaines propriétés pour les systèmes de communications exprimées sous la forme d'obligations de preuve pour les fonctions représentant les différents composants du réseau. Ces théorèmes sont démontrés pour toute instanciation du modèle pour un réseau donné.

Dans le modèle, le temps est implicite. Le temps ne progresse pas, chaque appel à la fonction fait progresser le système d'un pas. Un pas n'est pas vraiment une unité du temps. Il s'agit d'un pas de calcul pour le système : le calcul des routes et l'ordonnancement.

Parmi les travaux à venir, nous envisagerons de changer la façon dont le temps est vu dans GeNoC : calculer les routes et les étapes prochaines à chaque nœud et faire progresser le temps chaque fois, de façon identique à une simulation. La liste des voyages retardés sera l'entrée de l'étape suivante de l'exécution du modèle, avec une liste de tentatives modifiée.

La modélisation faite ici se place à un très haut niveau d'abstraction, algorithmique. Une extension sera de faire le lien entre ce modèle et une représentation du système au niveau RTL (Register Transfer Level – transfert de registres).

L'introduction du temps sera une étape fondamentale dans cette direction. Il existe déjà des travaux effectués pour passer du niveau RTL vers la logique des démonstrateurs des théorèmes. Ceci peut être utilisé comme une automatisation de la modélisation.

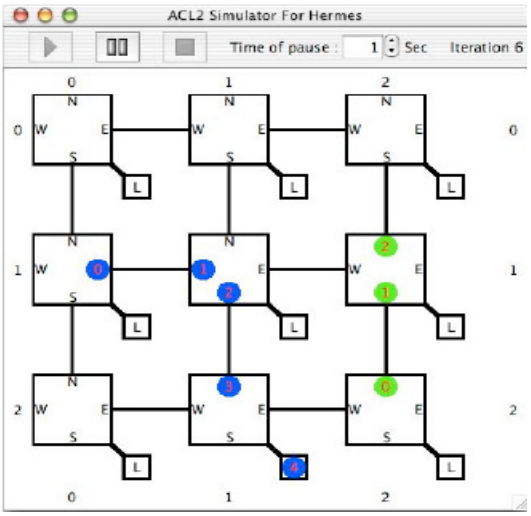

Fig. 15, interface de visualisation

BIBLIOGRAPHIE :

[1] J. Markoff, “Circuit Flaw Causes Pentium to Miscalculate: Intel Admits”, in New York Times, 24 Novembre 1994.

[2] R. Dubey: "Elements of Verification", SOC Central: White paper, Mars 25, 2005.

[3] J. Schmaltz: "Une formalisation fonctionnelle des communications sur la puce", thèse de Doctorat de l’Université Joseph Fourier, Janvier 2006.

[4] D. Borrione, A. Helmy, L. Pierre: “ACL2-based Verification of the Communications in the Hermes Network on Chip“, Proc. SMACD’06.

[5] http://www.cs.utexas.edu/users/moore/acl2/

[6] M. Kaufmann, P. Manolios, J Moore: "Computer Aided Reasoning: an Approach", Kluwer Academic Press, 2002.

[7] B. Brock, M. Kaufmann, J Moore: "ACL2 Theorems about Commercial Microprocessors", Proc. FMCAD'96, 1996.

[8] J Moore, T. Lynch, M. Kaufmann: "A Mechanically Checked Proof of the Correctness of the Kernel of the AMD5K86 Floating-Point Division Algorithm", IEEE Trans. on Computers, 47(9), Septembre 1998.

[9] M.Kauffman, P.Manolios, J.Moore: “Computer Aided Reasoning: ACL2 Case Studies”, Kluwer Academic Press, 2000.

[10] Radu Marculescu: "Energy, Fault-Tolerance, and Scalability Issues in Designing Network-on-Chip", tutorial at ASP-DAC'04, Yokohama (Japan), Janvier 2004.

[11] M. Schäfer, T. Hollstein, H. Zimmer, M. Glesner: "Deadlock-free routing and Component placement for irregular mesh-based networks-on-chip”, Proc. ICCAD 2005, San Jose, Novembre 2005, IEEE Computer Society.

[12] P. P. Pande, C. Grecu, A. Ivanov, R. Saleh, G. De Micheli: "Design, Synthesis, and Test of Networks on Chips", IEEE Design & Test, Vol. 22, issue 5, Septembre 2005.

[13] A. Jalabert, S. Murali, L. Benini, G. De Micheli : "XpipesCompiler: a tool for instantiating application specific Networks on Chip", Proc DATE'04, Février 2004.

[14] F. Moraes, N. Calazans, A. Mello, L. Möller, L. Ost: "HERMES: an infrastructure for low area overhead packet-switching networks on chip”, the VLSI Journal, vol. 38-1, 2004.

[15] http://toledo.inf.pucrs.br/~gaph/Projects/Hermes/Hermes.html

[16] W. J. Dally, B. Towles: "Principles and practices of interconnection networks", Elsevier, 2004.

[17] J. Schmaltz, D. Borrione: "A Generic Network on Chip Model", Proc. TPHOL'2005.

[18] J Moore: "Symbolic Simulation: An ACL2 Approach", Proc. FMCAD'98, Springer-Verlag LNCS 1522, Novembre 1998.

[19] P. Curzon, "Experiences Formally verifying a network component”, in Proc. 9th annual IEEE Conference on Computer Assurance. IEEE press, 1994.

[20] R. Bharadwaj, A.Felty, F.Stomp, “Formalizing inductive proofs of network algorithms,” in Proc. 1995 Asian Computing Science Conference, 1995.

[21] B. Gebremichael, F. Vaandrager, M. Zhang, K. Goossens, E. Rijpkema, and A. Radulescu, “Deadlock Prevention in the AEthereal Protocol,” in Proc. CHARME’05, Octobre 2005.

[22] E.M.Clarke, O.Grumberg, and S.Jha, “Verifying parameterized networks”, ACM Transactions on Programming Languages and Systems, vol. 19, no. 5, Septembre 1997.

[23] S.Creese and A.Roscoe, “Formal verification of arbitrary network topologies,” in Proc. PDPTA’99, 1999.